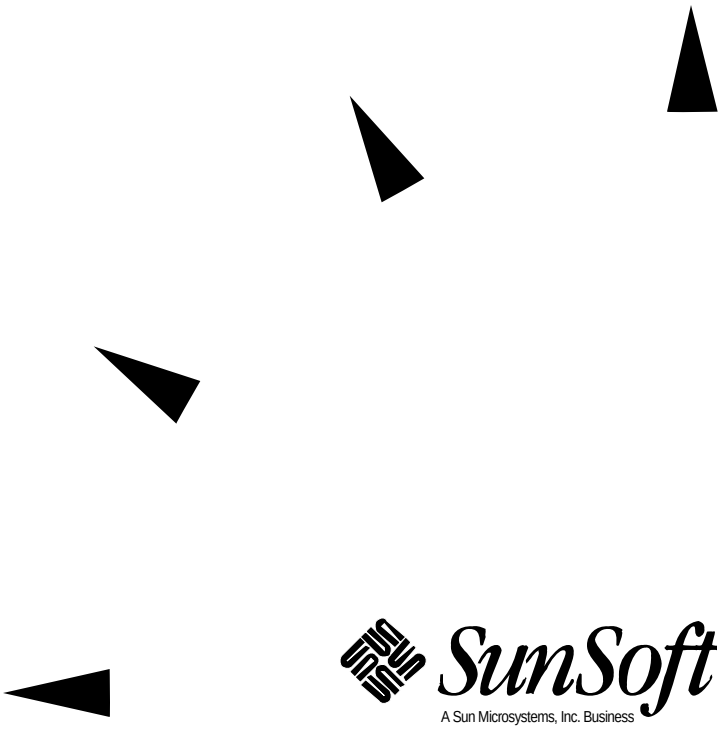
Solaris OpenGL 1.1 Implementation and Performance Guide



2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No: 805-1015-10
Revision A, July1997

SunSoft
A Sun Microsystems, Inc. Business

Please
Recycle

Contents

1. Introduction to the Solaris OpenGL Software 1

Overview 1

Solaris OpenGL 1.1 Product Functionality 1

Supported OpenGL 1.1 Extensions 2

Compatibility Issues 3

MT-Safe 3

Supported Platforms 4

Where to Look for Information on OpenGL Programming . . . 4

2. Solaris OpenGL Architecture 5

Acceleration vs. Optimization 5

A Quick Review of the OpenGL Architecture 6

Graphics Hardware Architecture 7

Solaris OpenGL Software Architecture 8

Vertex Processing Architecture 11

Rasterization and Fragment Processing Architecture 11

Solaris OpenGL Interface Layers 12

3. Performance 15

General Tips on Vertex Processing 15

Vertex Arrays 16

Consistent Data Types 16

Low Batching 17

Optimized Data Types 18

Creator3D Graphics and Creator Graphics Performance 19

Attributes Affecting Creator3D Performance 19

Attributes Affecting Creator Performance 26

Pixel Operations 30

GX Performance 33

4. X Visuals for the Solaris OpenGL Software 35

Programming With X Visuals for the Solaris OpenGL Software 35

Colormap Flashing for OpenGL Indexed Applications 38

GL Rendering Model and X Visual Class 38

Depth Buffer 38

Accumulation Buffer 39

Stencil Buffer 39

Auxiliary Buffers 39

Stereo 39

▼ To Set Up the Frame Buffer for Stereo Operation: 39

Rendering to DirectColor Visuals 40

Overlays 40

Server Overlay Visual (SOV) Convention. 40
Enabling SOV Visuals 41
OpenGL Restrictions on SOV. 41
Compatibility of SOV with other Overlay Models 42
Gamma Correction. 43
5. Tips and Techniques. 45
Identifying the Solaris OpenGL Library Version 45
Avoiding Overlay Colormap Flashing 46
Changing the Limitation on the Number of Simultaneous GLX Windows. 46
Hardware Window ID Allocation Failure Message. 47
Getting Peak Frame Rate. 47
Frequently Asked Questions 47

Figures

Figure 2-1 OpenGL Architecture 6

Figure 2-2 Solaris OpenGL Software Architecture 10

Figure 2-3 Solaris OpenGL Data Paths 13

Figure 3-1 Hardware Rasterizer Path for Creator3D 23

Figure 3-2 Software Rasterizer Data Path for Creator3d and Creator . . . 29

Figure 3-3 Solaris OpenGL Architecture for Drawing Pixels. 30

Tables

Table 2-1 Data Paths through the Solaris OpenGL System 9

Table 2-2 Performance Data for 3D Line Strips in Display List Mode . . 14

Table 2-3 Performance Data for 3D Triangle Strips in Display List Mode 15

Table 2-4 Performance Data for 3D Texture-Mapped Triangles in Display List Mode 15

Table 2-5 Performance Data for `glDrawPixels()` 16

Table 4-1 Visuals Available for Ultra Creator 3D 36

Preface

Solaris OpenGL 1.1 Implementation and Performance Guide provides information on the Solaris™ OpenGL™ 1.1 software.

Who Should Use This Book

This book is intended for application developers who are using the Solaris OpenGL software to port OpenGL applications to Sun hardware. It assumes familiarity with OpenGL functionality and with the principles of 2D and 3D computer graphics.

How This Book Is Organized

This book is organized as follows:

Chapter1," Introduction to Solaris OpenGL," provides a description of the Solaris OpenGL software.

Chapter 2, "Solaris OpenGL Architecture," presents information on the Solaris OpenGL architecture.

Chapter 3, "Performance," presents specific information on using Sun's OpenGL library for specific hardware platforms.

Chapter 4, "X Visuals for Solaris OpenGL," presents information on visuals for the Solaris OpenGL product.

Chapter 5, "Tips and Techniques," contains information that may make using the Solaris OpenGL library easier.

Related Books

For information on the OpenGL library, refer to the following books:

- Neider, Jackie, Tom Davis, Mason Woo, *OpenGL Programming Guide*, Reading, Mass., Addison-Wesley, 1993.
- OpenGL Review Board, *OpenGL Reference Manual*, Reading, Mass., Addison-Wesley, 1992.
- Kilgard, Mark, *OpenGL Programming for X Windows Systems*, Reading, Mass., Addison-Wesley, 1996.

What Typographic Changes Mean

The following table describes the typographic changes used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name% su` `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

Introduction to the Solaris OpenGL Software 1

Overview

The Solaris OpenGL software is Sun's native implementation of the OpenGL application programming interface (API). The OpenGL API is an industry-standard, vendor-neutral graphics library. It provides a small set of low-level geometric primitives and many basic and advanced 3D rendering features, such as modeling transformations, shading, lighting, anti-aliasing, texture mapping, fog, and alpha blending.

Solaris OpenGL 1.1 Product Functionality

The Solaris OpenGL 1.1 software is a functionally conforming implementation based on the OpenGL 1.1, GLX 1.2, and GLU 1.2 standard specifications. The Solaris OpenGL software incorporates the new features in OpenGL 1.1 and includes support for the `SERVER_OVERLAY_VISUALS` property.

OpenGL 1.1 Library

The OpenGL 1.1 library is a superset of OpenGL 1.0, including all OpenGL 1.0 functionality and additional features that were available as extensions to OpenGL 1.0. The added extensions, which are listed Table 1-1 on page 2, have become part of base OpenGL functionality; however, the semantics or syntax

may have changed for inclusion in OpenGL 1.1. For detailed information on the extensions incorporated into the OpenGL 1.1 specification, see Appendix C in *The OpenGL Graphics System: A Specification, Version 1.1*.

Table 1-1 OpenGL 1.1 Additions

OpenGL 1.1 Name	1.0 Extension Name	Changed Syntax or Semantics
Vertex arrays	`GL_EXT_vertex_array`	Yes
Polygon offset	`GL_EXT_polygon_offset`	Yes
RGBA logical operations	`GL_EXT_blend_logic_op`	No
Internal texture image formats	`GL_EXT_texture`	No
Texture replace environment	`GL_EXT_texture`	No
Texture proxies	`GL_EXT_texture`	Yes
Copy texture and subtexture	`GL_EXT_copy_texture` `GL_EXT_subtexture`	No
Texture objects	`GL_EXT_texture_object`	Yes

Note – Because the Solaris OpenGL 1.1 software is based on a more current version of the OpenGL specifications (OpenGL 1.1, GLX 1.2, GLU 1.2) than the Solaris OpenGL 1.0 version, Solaris OpenGL 1.0 customers should be alert for software changes required to support the updated OpenGL specifications.

Supported OpenGL 1.1 Extensions

The Solaris OpenGL 1.1 software supports the following OpenGL 1.1 extensions:

- 3D texture mapping extension – `GL_EXT_texture3D`
- ABGR reverse-order color format extension – `GL_EXT_abgr`
- Texture color table extension – `GL_SGI_texture_color_table`
- SGI color table extension – `GL_SGI_color_table`
- Sun geometry compression extension – `GL_SUNX_geometry_compression`
- Rescale normal extension - `GL_EXT_rescale_normal`

The Solaris OpenGL software also supports the following GLX extension:

- Return the transparent pixel index for an overlay/underlay window pair – `GLX_SUN_get_transparent_index`. See the `glXGetTransparentIndexSUN`(3gl) man page.

Note that other implementations of OpenGL may have used extensions that your application calls. To determine what extensions, if any, your application uses, search for command-name patterns such as `gl`*Procedure*`EXT(`3gl`)`. If your application uses extensions, you will need to ensure that it also handles the functionality in an OpenGL 1.1-compliant manner. To determine what extensions an OpenGL implementation supports, use `glXQueryExtensionString(`3gl).

Compatibility Issues

Applications compiled with the Solaris OpenGL 1.0 library will run unchanged with the Solaris OpenGL 1.1 implementation. However, note the following backward compatibility issues:

- To reduce function call overhead and improve performance for vertex calls in immediate mode, vertex commands such as `glVertex, glColor, glNormal, glTexCoord` and `glIndex` have been redefined as macros in the Solaris OpenGL 1.1 software. Therefore, by default, applications compiled with the Solaris OpenGL 1.1 library will not run on the 1.0 library. However, to compile an application with the Solaris OpenGL 1.1 library and maintain compatibility with 1.0, use the flag `-DSUN_OGL_NO_VERTEX_MACROS` when compiling the application. See the `glVertex` (3gl) man page for further information.
- If your application uses the new features in the Solaris OpenGL 1.1 library, it will not be backward compatible with the Solaris OpenGL 1.0 library.

MT-Safe

The Solaris OpenGL 1.1 library is MT-safe as long as there is only one OpenGL rendering thread. For example, an application can create an OpenGL rendering thread and a separate Xlib thread for handling a graphics user interface. To be MT-safe with Xlib, the application must call XInitThreads before any of the GLX calls.

1

Supported Platforms

The Solaris OpenGL 1.1 software supports the following devices:

- Creator Graphics and Creator3D Graphics – OpenGL functionality is accelerated in hardware.
- SX, ZX, GX, GX+, TGX, TGX+, S24 – OpenGL functionality is performed in software.
- All SMCC SPARC™ systems equipped with the following frame buffers are supported on the OpenGL 1.1 software: the TCX, SX, GX, ZX and Creator families of frame buffers. This includes Ultra™ desktop, Ultra Enterprise™ and all the legacy SPARCstation™ family.
- The software requirement is:
  - Solaris 2.5.1 plus patch 103796-09 or higher.
  - Solaris 2.6 software or higher.

Where to Look for Information on OpenGL Programming

For information on how to write an OpenGL application, see the following books:

- *OpenGL Programming Guide* by Neider, Davis, and Woo
- *OpenGL Reference Manual* by the OpenGL Architecture Review Board
- *OpenGL Programming for X Windows Systems* by Mark Kilgard

These books are published by Addison-Wesley and are available through your local bookstore.

For more information on OpenGL, you may want to refer to “The Design of the OpenGL Interface” written by Mark Segal and Kurt Akeley. A PostScript copy of this document is included in the `SUNWgldoc` package or the Solaris OpenGL 1.1 CD-ROM. For the complete specification of what constitutes OpenGL, see *The OpenGL Graphics System: A Specification, Version 1.1*, also written by Mark Segal and Kurt Akeley. An online version of this specification is located at `http://www.sgi.com/Technology/OpenGL/glspec1.1/glspec.html`).

Finally, for a good source of answers to questions you may have about OpenGL, see Silicon Graphics’s OpenGL information center at `http://www.sgi.com/Technology/OpenGL/opengl.html`.

Solaris OpenGL Architecture 2

The purpose of designing a graphics system architecture is to enable performance within the constraints of cost and functionality goals. Hardware design places various stages of the graphics pipeline into hardware accelerators. Software design uses the hardware features and complements the hardware by providing complete coverage of functionality.

Understanding the hardware and software architecture of a particular system will help you determine whether a feature is accelerated in the graphics hardware or implemented in software. This will enable you to identify which path through the system your application uses for the feature. With this information, you can project your application's performance. Given knowledge of performance versus functionality tradeoffs, you can make informed choices about how to use the system to maximize your application's interactivity.

This chapter describes the Solaris OpenGL architecture. First it defines two terms commonly used when discussing hardware and software performance.

Acceleration vs. Optimization

When discussing performance, understanding how the hardware implementor, software implementor, and application programmer define and differentiate the terms *hardware acceleration* and *software optimization* is helpful.

- To the hardware designer, hardware accelerating OpenGL means implementing logic in the form of gates and data paths for OpenGL functions.

- To the OpenGL software implementor, accelerating OpenGL functions means writing software to use the graphics hardware features. In addition, the software implementor can *optimize* OpenGL features that are not accelerated in hardware by writing highly tuned code to make the performance of those features as efficient as possible.
- To the OpenGL application programmer, acceleration typically means the speed at which various combinations of geometry and OpenGL state render, with the goal generally being interactive performance.

With these definitions in mind, the next sections describe the OpenGL architecture and at the implementation of this architecture in the Solaris OpenGL software.

A Quick Review of the OpenGL Architecture

As a first step in examining the Solaris OpenGL architecture, Figure 2-1 shows the basic architecture of the OpenGL library.

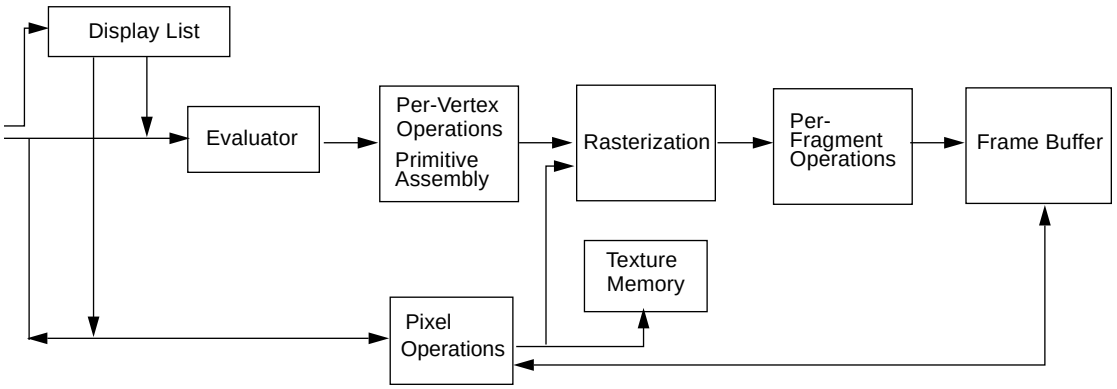


Figure 2-1 OpenGL Architecture

In the first stage of the OpenGL pipeline, vertex data enters the pipeline, and curve and surface geometry is evaluated. Next, colors, normals, and texture coordinates are associated with vertices, and vertices are transformed and lit. Vertices are then assembled into geometric primitives.

1. From Segal, Mark, and Kurt Akeley, "The OpenGL Graphics System: A Specification," Mountain View, CA, 1995.

The rasterization stage converts geometric primitives into frame buffer addresses and values, or fragments. Each fragment may be altered by per-fragment operations, such as blending. Per-fragment operations store updates into the frame buffer based on incoming and previously stored Z values (for Z buffering), blending of incoming fragment colors with stored colors, as well as masking and other logical operations.

Pixel data is processed in the pixel operation stage. The resulting data is stored as texture memory, or rasterized and processed as fragments before being written to the frame buffer.

The task of the hardware and software implementors at Sun was to implement the OpenGL functionality. The remainder of this chapter describes this implementation.

Graphics Hardware Architecture

Graphics hardware architectures can be designed to meet varying constraints of cost and CPU performance. High-performance model coordinate (MC) devices typically implement vertex processing and transformations in hardware. A model coordinate device may perform lighting, coordinate transformations, clipping, and culling as well as rasterization and fragment processing in hardware, thereby providing very fast performance.

At a different performance level, rasterization devices typically use the host CPU to perform vertex processing and use the rasterization hardware to convert device coordinate geometry into pixel values. The Ultra Creator and Creator3D systems are examples of device coordinate (DC) devices. The graphics hardware architecture of the Creator3D graphics system is designed as follows:

- Primitive assembly and vertex processing are performed on the UltraSPARC™ CPU. Texturing operations are also performed on the CPU.
- Rasterization and fragment processing are performed in the Creator3D Graphics hardware subsystem. The Creator3D graphics system accelerates rasterization of lines, points, and triangles, and also accelerates per-fragment operations such as the pixel ownership test, scissor test, depth buffer test, blending, logical operations, line anti-aliasing, line stippling, and polygon stippling.

The benefit of building custom hardware for graphics is that when operations are parallelized in hardware circuits, turning on features (like both Z-buffering and blending) has a very small performance cost. If a feature is provided in hardware, the hardware is usually designed to allow sustained throughput for that feature. Thus, you can make full use of features that have been implemented in hardware without experiencing performance degradation.

The benefit of putting graphics functions in software is that since the CPU is a required and shared computing resource, using it for graphics operations imposes no additional financial cost. The disadvantage is that each additional graphics operation requires CPU cycle time. When an application asks more of the CPU, the CPU may perform more slowly.

Solaris OpenGL Software Architecture

Once the hardware designers have determined what the hardware will accelerate, all other decisions regarding performance fall to the software implementors. Software implementors need to consider the following questions:

1. What hardware features will be used?
2. What features that are not accelerated in hardware can the software optimize?
3. How will the software implement all functionality?

In response to these questions, the Solaris OpenGL software developers implemented OpenGL as follows:

- Accelerated OpenGL by using using all features of the Creator and Creator3D graphics subsystems.
- For the Creator and Creator3D systems optimized line and point transformation and clip test, and a subset of texture lookup and filtering.
- Implemented OpenGL to its complete specification by writing code for primitive assembly and vertex processing, including:
  - Coordinate transformations
  - Texture coordinate generation
  - Clipping

- Implemented two forms of software rasterization for OpenGL features not rasterized in hardware:
  - Optimized software rasterizer for many texturing functions and pixel operations. Software rasterization is done by the CPU using an optimized implementation. On an UltraSPARC CPU, some features, such as texturing rasterization, may be handled using software code employing the VIS instruction set.
  - A software rasterizer for all features not handled by the hardware or by the VIS software.

This implementation of the Solaris OpenGL library allows devices with varying capabilities to run efficiently on the OpenGL software. It enables Solaris OpenGL applications to run on the following types of devices:

- Model coordinate device – Handles most OpenGL functionality in hardware, including vertex processing, primitive assembly, rasterization, and fragment operations.
- Device coordinate device (Creator or Creator3D graphics system) – Performs vertex processing. Rasterization and fragment processing is handled in hardware.
- Memory mappable devices (SX, ZX, GX, GX+, TGX, TGX+, TCX) – Vertex processing, primitive assembly, rasterization, and fragment processing are performed in software, and the results are written to the memory-mapped frame buffer.

Figure 2-2 on page 10 illustrates the graphics software architecture of the Solaris OpenGL product. This figure shows the paths that application data can take through the OpenGL system, depending on the type of hardware device the application is running on. Table 2-1 summarizes the data paths with reference to several hardware platforms.

Table 2-1 Data Paths Through the Solaris OpenGL System

Platform	Vertex Processing	Rasterization	Performance
MC device	Hardware vertex processing	Hardware rasterizer	Fastest path
	Software vertex processing	Hardware rasterizer	Fast path
	Software vertex processing	Software rasterizer	Slow path

Table 2-1 Data Paths Through the Solaris OpenGL System *(Continued)*

Platform	Vertex Processing	Rasterization	Performance
DC device (Creator3D or Creator)	Software vertex processing	Hardware rasterizer	Fast path
	Software vertex processing	Software rasterizer	Slow path
Memory map (ZX, GX, SX)	Software vertex processing	Software rasterizer	Only path

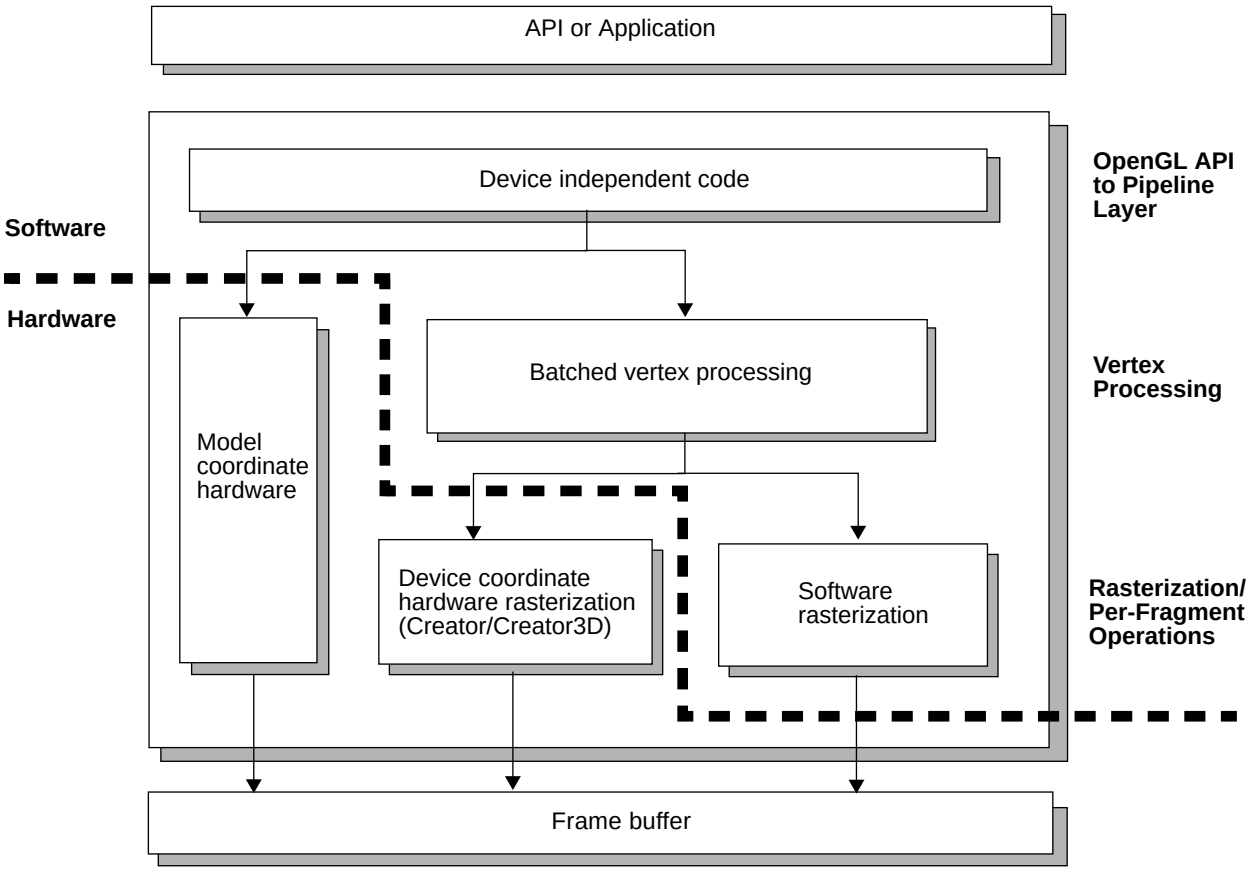


Figure 2-2 Solaris OpenGL Software Architecture

Vertex Processing Architecture

As Figure 2-2 shows, Sun's OpenGL implementation handles vertex processing in several ways:

- Hardware vertex processing – On model coordinate devices, vertex processing is done via the hardware. In addition to hardware acceleration, the model coordinate (MC) pipeline is optimized for vertex arrays and display list mode. The model coordinate pipeline also recognizes consistent data types within `glBegin`/`glEnd` pairs. If the data is consistent, the software is able to use hardware resources efficiently.
- Software vertex processor – This is the fully optimized path from the software implementor's point of view. The principal optimization is that the model coordinate software pipeline recognizes consistent data types within `glBegin`/`glEnd` pairs: if the data is consistent, the software pipeline is able to use CPU resources efficiently.

The OpenGL vertex array commands result in the best performance for vertex processing on all hardware platforms. For repeated rendering of the same geometry, display lists provide significant performance benefits over immediate mode rendering.

Rasterization and Fragment Processing Architecture

Rasterization and fragment processing is handled in one of the following ways:

- Hardware rasterizer – The graphics subsystem handles lines, points, and triangles, and does simple fragment processing, such as blending and the depth-buffer test.
- Optimized software rasterizer – The CPU does software rasterization using an optimized implementation. On an UltraSPARC CPU, some features, such as texturing rasterization, may be handled by the UltraSPARC CPU using software code employing the VIS instruction set.
- Software rasterizer – The CPU does software rasterization using a generic, unoptimized implementation. The generic software rasterizer is approximately one-sixth the speed of the optimized software rasterizer.

2

Solaris OpenGL Interface Layers

The Solaris OpenGL implementation has three layers of interfaces with the hardware, each requiring successively more processing by the host CPU. These interface layers correspond to the stages of the OpenGL pipeline. The rendering interface is determined by the value of the current OpenGL attributes, and in a small number of cases by the geometry itself. In general, the more host processing needed, the slower the resulting rendering, so an application should avoid attributes that force the slower rendering layers to be used.

Figure 2-3 on page 13 shows the interface layers and their relationship to data paths through the Solaris OpenGL system. In this illustration, the filled boxes represent the hardware-specific device pipeline (DP) components and show the hardware data paths. The white boxes represent the device-independent (DI) software components and show the software data paths.

The more efficiently an application can reach a filled box, the better the application's performance will be. For example, for an application running on a model coordinate device, the fast data paths are those that result in rendering in hardware at the vertex processing layer. Setting an attribute that causes the use of the software pipeline for model coordinate processing can result in a significant drop in performance. Setting an attribute that results in the use of software rasterizing can cause an even more significant drop in performance.

On a device coordinate device such as the Creator3D system, hardware rasterization is about three times faster than the VIS (optimized) rasterizer. The VIS rasterizer is about five-to-six times faster than the generic software rasterizer. Thus, the best way to increase rasterization and fragment processing performance on a DC device is to stay in the hardware rasterizer whenever possible.

Memory-mappable devices without hardware support use the software pipeline for model coordinate operations and the software rasterizer for rasterization. Examples of this device are the single-buffered GX, and TGX. For devices that do not allow memory access, the Solaris OpenGL architecture provides a pixel--rendering interface layer. However, at this time no Sun hardware devices use this interface layer.

For detailed information on attributes that result in slower rendering paths, see Chapter 3, "Performance."

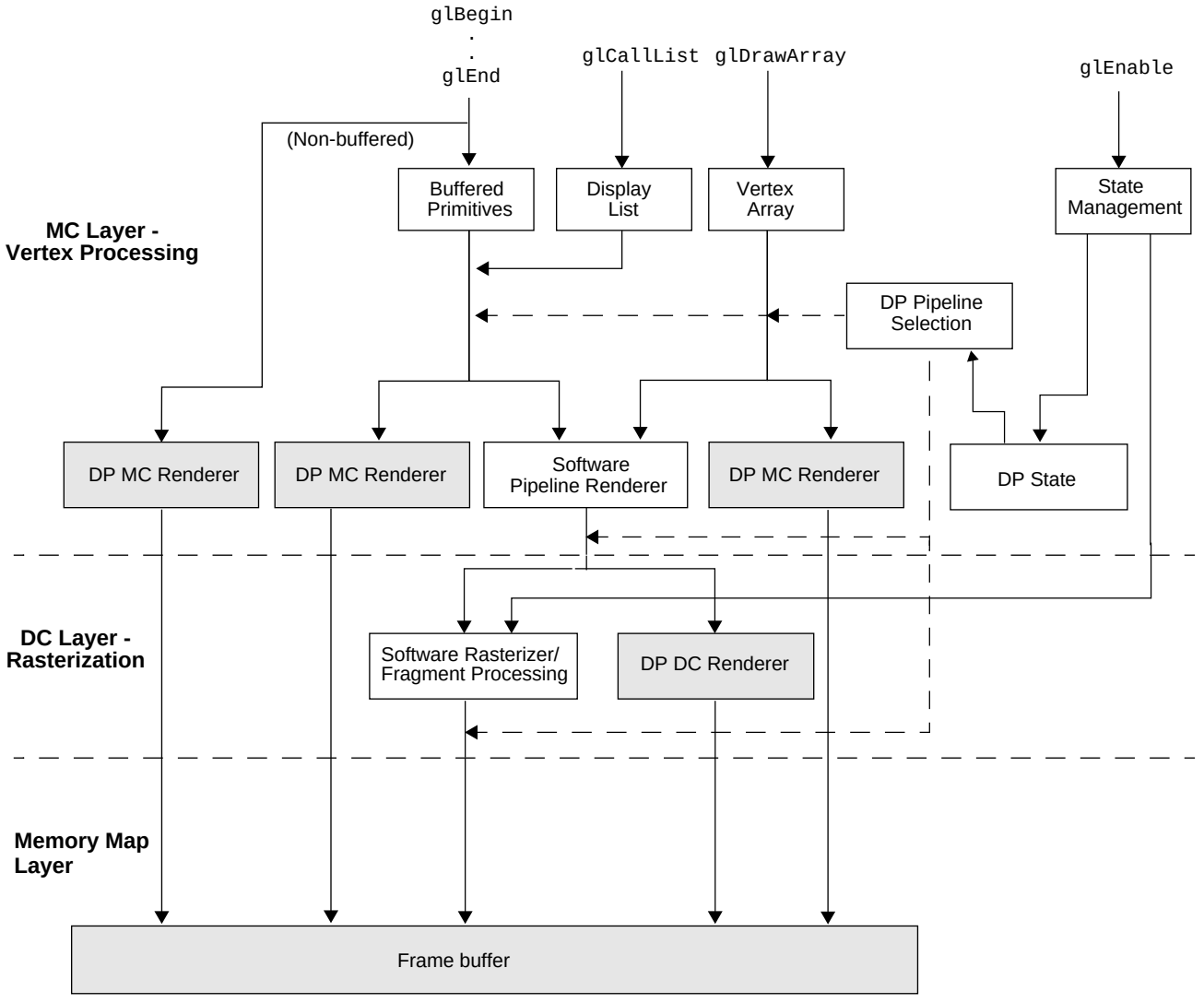


Figure 2-3 Solaris OpenGL Data Paths

2

Performance 3

This chapter provides performance information that you can use to tune your application to make the best use of Sun hardware graphics accelerators. The first section provides general advice on how to optimize vertex processing performance for a variety of platforms. The subsequent sections provide specific techniques to ensure maximum performance on the Creator3D and Creator graphics accelerators.

General Tips on Vertex Processing

To achieve the best vertex processing performance on all Sun platforms, follow these guidelines:

1. Use vertex arrays or display list mode rather than immediate mode whenever rendering data repeatedly.
2. Use consistent patterns of data types between `glBegin`(3gl) and `glEnd`(3gl). Consistent data types are described in “Consistent Data Types” on page 16.
3. If you must use immediate mode, try to include as many primitives of the same type as possible between one `glBegin` and the corresponding `glEnd`.
4. If vertex array is used, try to stay in vertex array mode, rather than switching between vertex array and immediate mode.

These guidelines are discussed in the sections that follow.

3

Vertex Arrays

Vertex array commands provide the best performance for vertex processing of big primitives because they avoid the function call overhead of passing one vertex, color, and normal at a time. Instead of calling an OpenGL command for each vertex, you can pre-specify arrays of vertices, colors, and normals, and use them to define a primitive or set of primitives of the same type with a single command. Interleaved vertex arrays may enable even faster performance, since the application passes the data packed in a single array.

Consistent Data Types

For the Solaris OpenGL implementation on all Sun platforms, vertex processing is optimized if the application provides consistent, supported data types within a `glBegin`/`glEnd` pair. Data types are consistent when the commands between one vertex call, such as `glVertex3fv`, and the next vertex call include identical patterns of data types in the identical order. In other words, consistent data is data for which the pattern is the same for each vertex, except when `glCallList` or `glEval*` is included. For example, the following set of commands is consistent because the primitive is defined by the repeating set of calls `glColor3fv`(3gl); `glVertex3fv`(3gl).

```
glBegin(GL_LINES);
   glColor3fv(...);
   glVertex3fv(...);
   glColor3fv(...);
   glVertex3fv(...);
   glColor3fv(...);
   glVertex3fv(...);
glEnd();
```

As another example, the following set of commands is consistent since each vertex contains the same data- a color, normal, and vertex in repeating order.

```
glBegin(GL_LINES);
   glColor3f(...);
   glNormal3f(...);
   glVertex3f(...);
   glColor3f(...);
   glNormal3f(...);
   glVertex3f(...);
glEnd();
```

Note – The `*f` versions of the calls may be used interchangeably with the `*fv` versions.

Inconsistent data types do not follow a repeating, supported pattern. In the first example below, the data is inconsistent because the first vertex has a normal, but the second vertex doesn't. In the second example, the order is reversed in the second set of commands, although both vertices have a color and a normal.

```
glBegin(GL_LINES);
   glNormal3fv(...);
   glColor3fv(...);
   glVertex3fv(...);
   glColor3fv(...);
   glVertex3fv(...);
glEnd();

glBegin(GL_LINES);
   glColor3fv(...);
   glNormal3fv(...);
   glVertex3fv(...);
   glNormal3fv(...);
   glColor3fv(...);
   glVertex3fv(...);
glEnd();
```

For general information on the vertex data that can be specified between `glBegin`(3gl) and `glEnd`(3gl) calls, see the `glBegin`(3gl) reference page.

Low Batching

Solaris OpenGL 1.1 performs best when given big primitives. If small primitives are sent to the library, the library will try to batch these primitives together, providing that the primitives are of the same primitive type, with the same consistent data pattern, and there are no attribute state changes outside the `glBegin` call.

For example, the following primitives will be batched together by the library.

```
glBegin(GL_TRIANGLES);
   glNormal3fv(...);
   glVertex3fv(...);
```

```
   glNormal3fv(...);
   glVertex3fv(...);
   glNormal3fv(...);
   glVertex3fv(...);
glEnd();

glBegin(GL_TRIANGLES);
   glNormal3fv(...);
   glVertex3fv(...);
   glNormal3fv(...);
   glVertex3fv(...);
   glNormal3fv(...);
   glVertex3fv(...);
glEnd();
```

The following example shows that the primitives are not batched together because the `glColor3fv` call outside the `glBegin` call breaks the batching of the primitives.

```
glBegin(GL_LINES);
   glVertex3fv(...);
   glVertex3fv(...);
glEnd();

glColorfv(...);
glBegin(GL_LINES);
  glVertex3fv(...);
  glVertex3fv(...);
glEnd();
```

Optimized Data Types

On any platform that uses the software pipeline for model coordinate rendering, your application will get better performance if it can pass vertex data in patterns for which the software pipeline has optimized code. Optimized data patterns are consistent data patterns which contain none of the following:

- glEdgeFlag*()
- glMaterial*()
- glEvalCoord*()

- glCallList() or glCallLists()
- both glColor*() and glIndex*()
- both glTexCoord*() and glIndex*()

Creator3D Graphics and Creator Graphics Performance

The Ultra Creator and Creator 3D Graphics systems accelerate rasterization of lines, points, and triangles as well as most per-fragment operations. Vertex processing and texturing operations are performed on the UltraSPARC CPU. The Solaris OpenGL implementation for the Creator and Creator3D frame buffers uses all features of the Creator graphics subsystem.

Rasterization and fragment processing is handled in one of three ways:

- Creator3D hardware rasterizer – Handles lines, points, and triangles, and does simple fragment processing.
- Optimized software rasterizer – UltraSPARC VIS (Visual Instruction Set) handles many texturing functions and pixel operations.
- Generic software rasterizer – Performs rasterization for all features not handled by the hardware or by the VIS software.

To find out more about the Creator and Creator3D hardware platforms, refer to the Architecture Technical White paper at `http://www.sun.com/desktop/products/Ultra2/`.

The following sections provide specific information on attribute use and pixel operations on these platforms.

Attributes Affecting Creator3D Performance

Primitive-attribute settings affect performance; therefore, you will get a better level of performance if you can avoid setting the attributes listed below. In some cases, the listed attributes simply increase the amount of processing in the hardware or optimized software data paths. In other cases, setting these attributes forces the use of the software rasterizer, resulting in slow performance.

Attributes That Increase Vertex Processing Overhead

Attributes that that result in more vertex processing overhead include:

- Enabling lighting.
- Turning on user specified clip planes (`GL_CLIP_PLANE[i]`).
- Enabling color material (`GL_COLOR_MATERIAL`).
- Enabling non-linear fog (`glFog(GL_FOG_MODE, GL_EXP{2})`). An exception to this is using `RGBA` mode on Creator3D Series `2`.
- Enabling `GL_NORMALIZE`.
- Turning on polygon offset. However, polygon offset is optimized for the case when the factor parameter of the `glPolygonOffset` call is set to 0.0. Users may have to adjust the units parameter accordingly to avoid stitching for this case.

Primitive Types and Vertex Data Patterns That Increase Vertex Processing Overhead

Types and patterns that result in more vertex processing overhead are:

- Using a surface primitive type as an argument to `glBegin`. The surface primitive types are: `GL_TRIANGLES`, `GL_TRIANGLE_STRP`, `GL_TRIANGLE_FAN`, `GL_QUADS`, `GL_QUAD_STRIP` and `GL_POLYGON`.
- Using a vertex data pattern for `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, and `GL_LINE_LOOP`, *other than* one of the following repeating patterns. These are the patterns that are maximally accelerated.

  V3F:

  ```
  glVertex3f(...);
  ...
  ```

  C3F_V3F:

  ```
  glColor3f(...);
  glVertex3f(...);

  ...
  ```

C4F_V3F:

```
glColor4f(...);
glVertex3f(...);

...
```

V2F:

```
glVertex2f(...);
...
```

C3F_V2F:

```
glColor3f(...);
glVertex2f(...);

...
```

C4F_V2F:

```
glColor4f(...);
glVertex2f(...);

...
```

- Using `glDrawElements` in immediate mode.

Attributes That Increase Hardware Rasterization Overhead

Attributes that result in slower hardware rasterization are:

- Enabling line antialiasing (`GL_LINE_SMOOTH`)
- Enabling point antialiasing (`GL_POINT_SMOOTH`)

Attributes That Force the Use of the Software Rasterizer

Setting the following attributes forces the use of the software rasterizer. This is the slowest data path. If your application requires any of the following attributes for performance critical functionality, you may want to determine whether this performance is acceptible. If not, you can evaluate whether the use of these attributes is advisable.

1. Rasterization attributes
   - In Indexed color mode, enabling line anti-aliasing (`GL_LINE_SMOOTH`) or point anti-aliasing (`GL_POINT_SMOOTH`)
   - Enabling polygon anti-aliasing (`GL_POLYGON_SMOOTH`)
   - Stippled lines (`GL_LINE_STIPPLE`) where the line stipple scale factor is larger than 15
   - Non-antialiased (“jaggy”) points with `glPointSize`(3gl)greater than 1.0

Note – The only anti-aliased point size supported by Creator3D and Creator is 1.0. `glPointSize` is ignored for anti-aliased points. Although the nominal antialiased point size is 1.0, the actual visible size is approximately 1.5.

---

2. Fragment Attributes
   - Blending (`GL_BLEND`) forces the use of the software rasterizer unless both the source and destination blend functions are in the following set of blend functions supported by the hardware:
     `GL_ZERO`
     `GL_ONE`
     `GL_SRC_ALPHA`
     `GL_ONE_MINUS_SRC_ALPHA`
   - Enabling the stencil test (`GL_STENCIL_TEST`)

     On the UltraSPARC platform, a VIS optimized software rasterizer is used for smooth-shaded non-textured stenciled triangles whenever the `glStencilOp` parameter *fail* is anything other than `GL_INCR` or `GL_DECR` and the depth test does not affect the stencil buffer. (This is the case when depth test is disabled or the `glStencilOp` parameters *zfail* and *zpass* are identical).
   - Enabling any type of fog in Indexed color mode

Figure 3-1 shows the data path for hardware rasterization on the Creator3D system. Figure 3-2 on page 29 illustrates the data path that the application uses when it sets an attribute that forces the use of the software rasterizer.

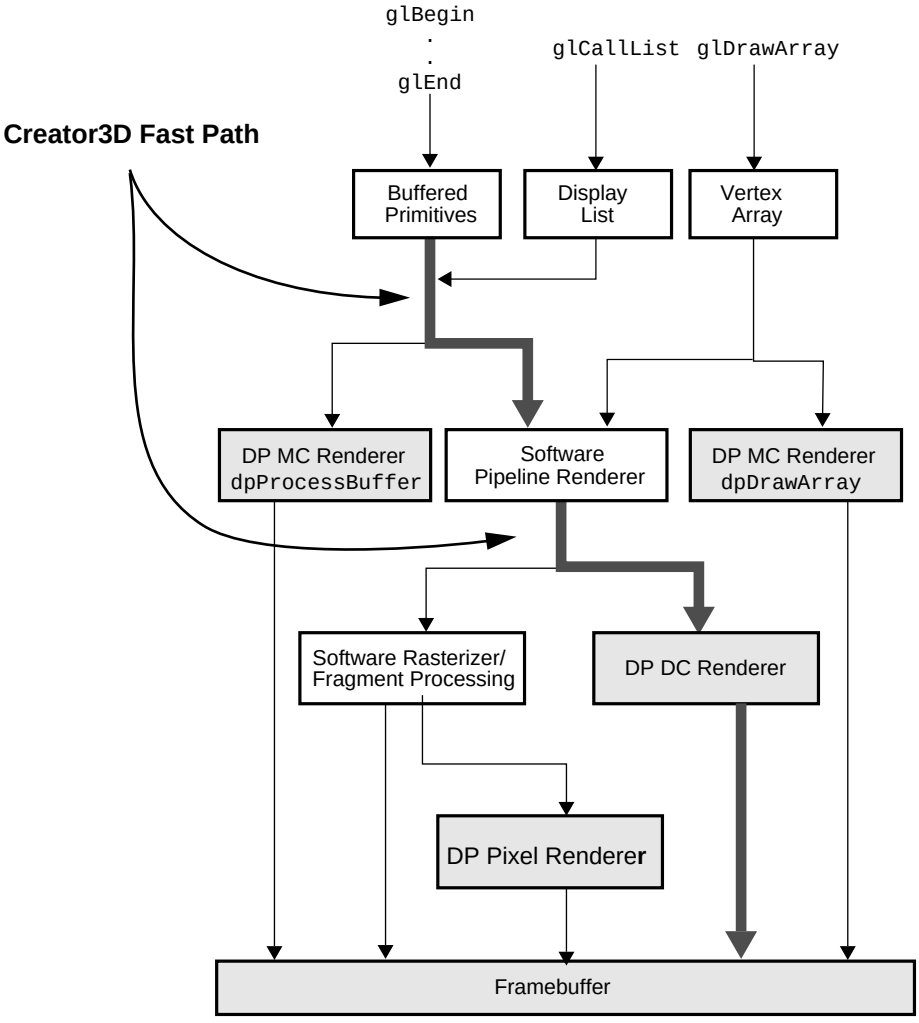


Figure 3-1 Hardware Rasterizer Path for Creator3D

3. Texturing Attributes

- Color Table -- When the GL_TEXTURE_COLOR_TABLE_SGI extension is used, the only glTexEnv texture base internal formats that are accelerated are GL_LUMINANCE, GL_LUMINANCE_ALPHA and GL_INTENSITY.
  - The texture environment mode glTexEnv GL_TEXTURE_ENV_MODE of GL_BLEND is not accelerated when it is used with the GL_TEXTURE_COLOR_TABLE_SGI extension.
- Fog -- On Creator3D, only linear fog is accelerated. On Creator3D Series 2, all types of RGBA fog are accelerated.

Attributes That Vary Optimized Texturing Speed

The VIS optimized software rasterizer will vary in texturing speed based on the texturing attributes specified. The factors affecting texturing speed are listed below. Note that this is *variance within the optimized path*, not the difference between the optimized and generic paths.

- Projection Type -- The type of projection matrix. Orthographic is faster than perspective.
- Wrap Mode -- Best speed is when all dimensions (GL_TEXTURE_WRAP_*x*) are set to GL_REPEAT.
- Dimension -- In general, 2D texturing is faster than 3D texturing, since there is one less texture coordinate to deal with. However, this does not mean it is better to use many 2D textures to approximate 3D texturing since the texture load time (see next section) may significantly increase the overhead.
- Minfilter -- The fastest GL_TEXTURE_MIN_FILTER parameter is GL_NEAREST, which is approximately 4x the speed of GL_LINEAR. After that the approximate relative speed in decreasing order is: GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, and GL_LINEAR_MIPMAP_LINEAR.
- Magfilter -- For GL_TEXTURE_MAG_FILTER, the same speed ratio of 4x applies to GL_NEAREST vs. GL_LINEAR. Note, however, that GL_TEXTURE_MAG_FILTER is ignored when GL_TEXTURE_MIN_FILTER is set to GL_NEAREST or GL_LINEAR. This can be overridden with a shell environment variable but this will slow down texturing speed for GL_NEAREST and GL_LINEAR, since they now have to perform level-of-detail calculations to determine when to use GL_TEXTURE_MAG_FILTER. The shell environment variable that forces this slower behavior is:

```
setenv SUN_OGL_MAGFILTER “conformant”
```

- Interior Texture Coordinates -- Before a triangle is textured, the texture coordinates at the triangle's vertices are checked to determine if they are *all* at least 1/2 texel away from the texture map edges towards the inside of the texture. Triangles that pass this criterion are rendered faster than triangles whose texture coordinates touch or cross the texture map's edges. Note that since quads are broken up into two triangles before texturing that this applies to quads as well. It also applies to each primitive in a connected list such as tristrip or quadstrip.
- Env Mode -- The fastest `glTexEnv()` `GL_TEXTURE_ENV_MODE` is `GL_REPLACE`, followed closely by `GL_MODULATE`. `GL_DECAL` is the same speed as `GL_REPLACE`.
- Color Table -- The use of the extension `GL_TEXTURE_COLOR_TABLE_SGI` will reduce texturing speed.

Attributes That Vary Texture Load Time

The time to load the texture image into a texture object or a display list will vary depending on the pixel store and pixel transfer attributes specified when the texture is specified. The following recommendations should be followed where possible to reduce texture load time:

- If multiple textures are being used, put the textures in texture objects and use `glBindTexture` to switch among the textures. This enables the texture load operation to be performed only once.
- If for some reason texture objects cannot be used, then the next best thing is to put the texture into a display list, making sure to fully specify in the display list the scale and bias for `glPixelTransfer` that are used in the application. The intent is to not have the display list inherit any changes to its initial pixel transfer from the calling environment. This avoids reprocessing the texture image. Avoid calling `glPixelMap` and `glColorTableSGI` (with target `GL_COLOR_TABLE_SGI`) after creating the display list that contains the texture image. Avoid calling `glPixelMap` and `glColorTableSGI` (with target `GL_COLOR_TABLE_SGI`) inside the display list. Doing so will cause the texture image to be reprocessed for every `glCallList`.
- Avoid setting any of the `glPixelTransfer` parameters to anything other than their default values.

- For GL_RGBA textures, use the extension GL_ABGR_EXT to specify the texture format and GL_UNSIGNED_BYTE for the texture data type.
- For 3D textures, some combinations of base internal format and incoming texture image format are optimized as given in the table below. Note that these optimized cases are valid only for data type GL_UNSIGNED_BYTE.

Table 3-1 3D optimized cases

Format	Base Internal Format
GL_LUMINANCE_ALPHA	GL_LUMINANCE_ALPHA
GL_RED	GL_INTENSITY
GL_RED	GL_LUMINANCE
GL_ALPHA	GL_ALPHA
GL_LUMINANCE	GL_INTENSITY
GL_LUMINANCE	GL_LUMINANCE
GL_ABGR_EXT	GL_RGBA

Attributes Affecting Creator Performance

This section applies when pure software rendering is being used. This happens on the single-buffered Creator platform when glDrawBuffer(3gl) is set to GL_BACK or GL_FRONT_AND_BACK. The data presented here is also valid for the SX, ZX, GX, GX+, TGX, TGX+, and TCX platforms. Note that for non-Ultra machines, VIS rasterization is replaced by an optimized software rasterizer.

Attributes That Increase Vertex Processing Overhead

Attributes that result in more vertex processing overhead are:

- Enabling lighting.
- Turning on user specified clip planes (GL_CLIP_PLANE[i]).
- Enabling color material (GL_COLOR_MATERIAL).
- Enabling non-linear fog (glFog (GL_FOG_MODE, GL_EXP{2})). An exception to this is using RGBA mode on Creator3D Series 2.
- Enabling GL_NORMALIZE.

- Turning on polygon offset. However, polygon offset is optimized when the factor parameter of the `glPolygonOffset` call is set to 0.0. Users may have to adjust the units parameter accordingly to avoid stitching for this case.

Attributes That Force the Use of the Generic Software Rasterizer

Setting the following attributes forces the use of the generic software rasterizer. This is the slowest data path. If your application requires any of the following attributes for performance critical functionality, you may want to determine whether this performance is acceptable. If not, you can evaluate whether the use of these attributes is advisable.

1. Texturing Attributes
   - All three-dimensional texturing attributes result in the use of the generic software rasterizer.
   - Two-dimensional texture mapping (`GL_TEXTURE_2D`) in the following cases:
     a. Texture environment mode `glTexEnv GL_TEXTURE_ENV_MODE` is set to `GL_BLEND`.
     b. `glTexEnv` texture base internal format is `GL_ALPHA.`
     c. Texturing of points is handled by the generic software.
     d. Fog is enabled.
     e. Any use of the SGI Texture Color Table (`GL_SGI_texture_color_table`) extension.
2. Fragment Attributes
   - Enabling any type of fog in Indexed color mode.
   - Enabling blending (`glBlendFunc`) (3gl) except when the source blending factor is `GL_SRC_ALPHA` and the destination blending factor is `GL_ONE_MINUS_SRC_ALPHA`. This case is optimized.
   - Enabling logical operations.
   - Enabling depth test `glEnable(GL_DEPTH_TEST)` forces the use of the optimized software rasterizer. If depth test is enabled, then if `glDepthFunc(3gl)` is on, enabling any Z comparison other than `GL_LESS` or `GL_LEQUAL` forces the use of the generic software rasterizer.

- Enabling alpha test.
- Setting `glDrawBuffer(3gl)` to `GL_BACK` or `GL_FRONT_AND_BACK`, or setting `glReadBuffer(3gl)` to `GL_BACK`.

Index Mode

When pure software rendering is being used, index mode rendering is handled by the generic software rasterizer. This includes any logic operation, blending, fog, stencil, alpha test, and the above-mentioned cases for Z comparison.

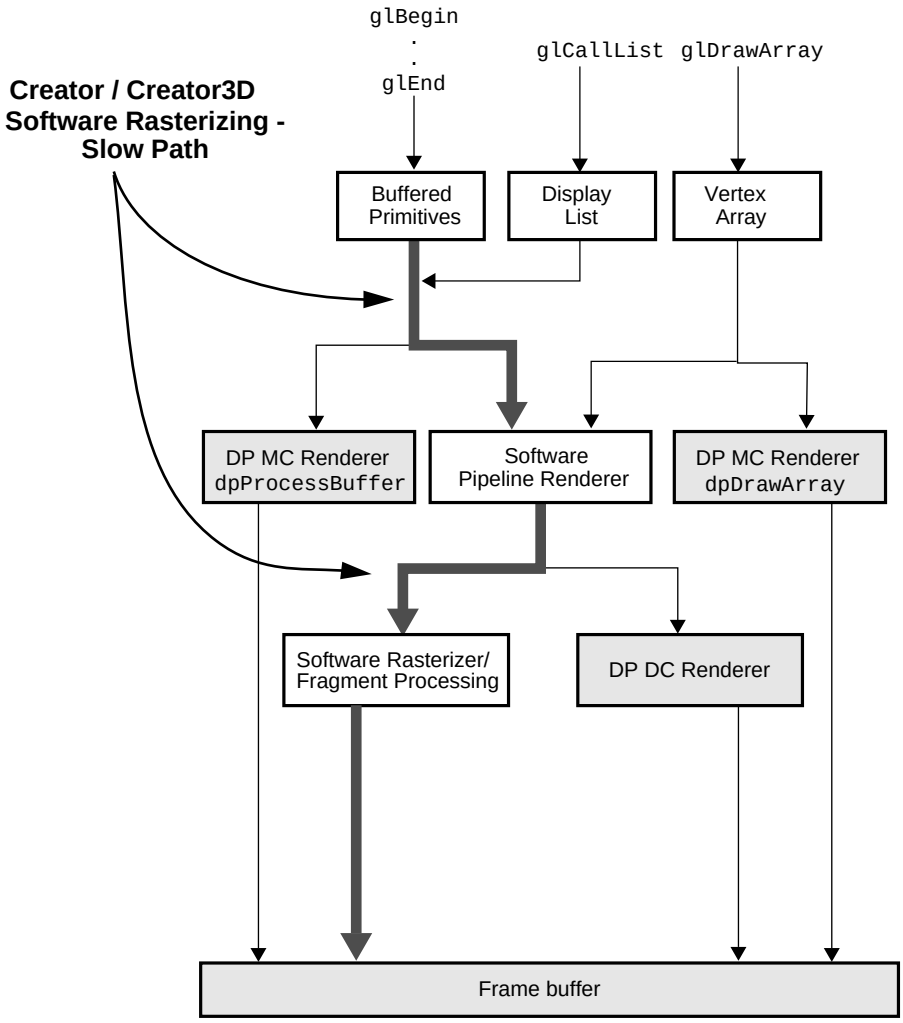


Figure 3-2 Software Rasterizer Data Path for Creator3d and Creator

Pixel Operations

Under optimal conditions, the commands `glDrawPixels(3gl)`, `glReadPixels`(3gl), and `glCopyPixels`(3gl) are optimized on the Creator and Creator3D systems using the VIS instruction set on the UltraSPARC CPU. Bitmap operations using the command `glBitmap`(3gl) are accelerated in the Creator3D font registers. However, some attribute settings result in the use of the software rasterizer for pixel operations.

Figure 3-3 shows the rasterization and fragment processing architecture for `glDrawPixels`(3gl). The figure shows the optimized and unoptimized paths for pixel rendering. Your application will experience performance degradation for each functional box that it needs. In addition, performance degradation will occur if the data type is not unsigned byte; in this case, the data must be reformatted internally.

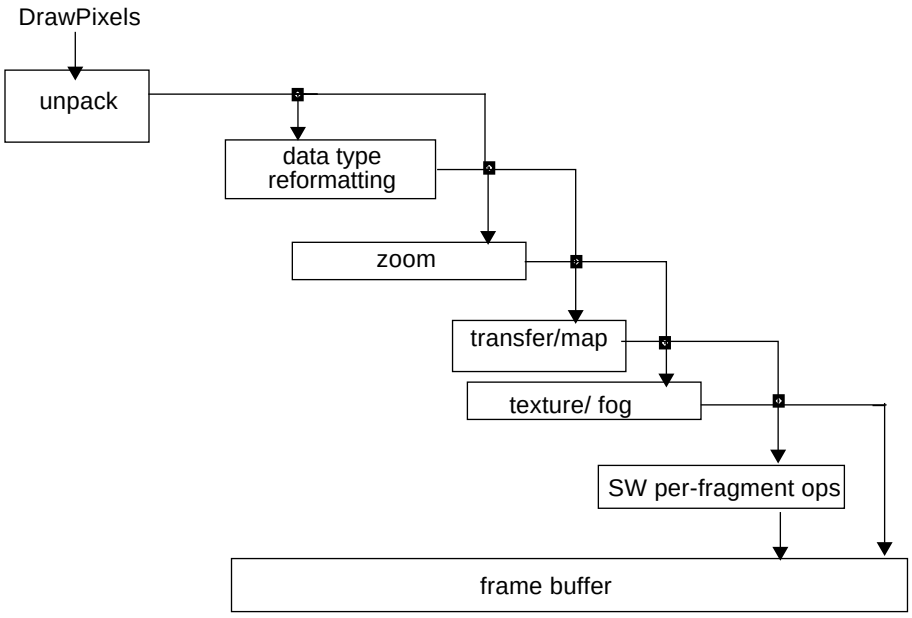


Figure 3-3 Solaris OpenGL Architecture for Drawing Pixels

Conditions That Result in VIS Optimization on Creator3D Systems

In general, for DrawPixels, CopyPixels, and Bitmap, the use of texture mapping or nonlinear fog (except in RGBA mode on Creator3D Series 2) will force the use of the generic software rasterizer, resulting in slow performance. In addition, if the hardware does not support the per-fragment operations that the application has enabled, the generic software rasterizer is used. See the OpenGL documentation or the "OpenGL Machine" diagram for a list of per-fragment operations.

For the Creator3D system, if the following conditions are true, pixel operations are optimized. If these conditions are not true, the generic software rasterizer is used.

`glDrawPixels` Command

- Pixel format is `GL_RGBA`, `GL_RGB`, `GL_ABGR_EXT`, `GL_RED`, `GL_GREEN`, `GL_BLUE` and `GL_LUMINANCE`.
- Data type is `GL_UNSIGNED_BYTE`.
- For the format of `GL_DEPTH_COMPONENT`, the types `GL_INT`, `GL_UNSIGNED_INT`, and `GL_FLOAT` are optimized.
- Texturing is disabled.
- Pixel unpacking is unnecessary.
- Pixel transfer, mapping, and zooming are in the default state.
- Fog mode is linear (Creator 3D) or any fog mode (Creator 3D Series 2).
- The fragment attributes are *not* those listed in "Fragment Attributes" on page 24.

`glReadPixels` Command

- Pixel format is `GL_RGBA`, `GL_RGB`, `GL_ABGR_EXT`, `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_LUMINANCE` and `GL_LUMINANCE_ALPHA`.
- Data type is `GL_UNSIGNED_BYTE`.
- For the format of `GL_DEPTH_COMPONENT`, the types `GL_INT`, `GL_UNSIGNED_INT`, and `GL_FLOAT` are optimized.
- Pixel packing is unnecessary.
- Pixel transfer and mapping are in the default state.

`glCopyPixels` Command

- Pixel type is `GL_COLOR`.
- Texturing is disabled.

- Pixel transfer, mapping and zooming are in the default state.
- Fog mode is linear (Creator 3D) or any fog mode (Creator 3D Series 2).
- The fragment attributes are *not* those listed in “Fragment Attributes” on page 24.

glBitmap(3gl)Command

- Texturing is not enabled.
- Blending is not enabled.

Conditions That Result in VIS Optimization on Creator Systems

For the Creator and non-Creator SMCC frame buffers, if the following conditions are true, pixel operations are optimized. If these conditions are not true, the generic software rasterizer is used.

glDrawPixels Command

- Pixel format is GL_RGBA, GL_RGB or GL_ABGR_EXT.
- Data type is GL_UNSIGNED_BYTE.
- Texturing is disabled.
- Pixel unpacking is unnecessary.
- If depth test is enabled, then if glDepthFunc(3gl) is on, enabling any Z comparison other than GL_LESS or GL_LEQUAL.

glReadPixels Command

- If glReadPixels format is GL_RGBA, GL_RGB or GL_ABGR_EXT, the pixel type GL_UNSIGNED_BYTE is optimized.
- If glReadPixels format is GL_DEPTH_COMPONENT, then these pixel types are optimized: GL_INT, GL_UNSIGNED_INT, or GL_FLOAT.
- Pixel packing is unnecessary.

glCopyPixels Command

- Pixel type is GL_COLOR.
- Texturing is disabled.
- Enabling any Z comparison other than GL_LESS or GL_LEQUAL.

glBitmap Command

- Texturing is disabled.

- If depth test is enabled, then if `glDepthFunc` is on, enabling any Z comparison other than `GL_LESS` or `GL_LEQUAL`.

GX Performance

GX performance is affected by attributes that force the use of the generic software rasterizer:

1. Texturing Attributes
    a. Only triangles are optimized. Texturing of points and lines is handled by the generic software.
    b. Texture environment mode `glTexEnv`(3gl) `GL_TEXTURE_ENV_MODE` is `GL_BLEND`.
2. Fragment Attributes
    a. Stencil operations
    b. Logic operations
    c. Any blending operation
    d. Linear or nonlinear fog
    e. Enabling any Z comparison other than `GL_LESS` or `GL_LEQUAL`

3

X Visuals for the Solaris OpenGL Software 4

Programming With X Visuals for the Solaris OpenGL Software

OpenGL rendering is supported on a subset of the visuals exported by the Solaris X window server on the Creator and Creator3D workstations. Because GLX overloads the core X visual classes with a set of attributes that indicate frame buffer capabilities, such as double buffer mode or stereo capabilities, the number of visuals supported by an OpenGL-capable X server is potentially large. For example, for the 24-bit TrueColor visual, the Solaris X window server on the Creator and Creator3D workstations exports the following types of GLX visuals: double buffer, single buffer, monoscopic, and stereoscopic.

This approach of exporting multiple GLX visuals for each X protocol core visual is colloquially referred to as the GLX *expansion* (or *visual explosion)*. For each different type of GLX visual that is exported, there is a corresponding X protocol core visual. Thus, there are multiple GLX visuals whose core X visual attributes are all identical.

Note – Solaris OpenGL 1.1 does not support windows with backing store. Enabling backing store on a window will penalize the user's Creator3D rendering performance.

Various OpenGL-capable visuals are supported in various releases of the Solaris operating environment. These are the visuals that an OpenGL program can use. This information applies to both Creator3D and Creator systems.

- In the Solaris 2.5.1 release, there are no expanded visuals. Expanded visuals are not supported in this release.
- In Solaris 2.5.1-based systems, expanded visuals are *disabled* by default. The user will have the option of enabling or disabling expanded visuals by using the command `ffbconfig -expvis` <*enable*|*disable*>.

See Table 4-1 and Table 4-2 for detailed information on using OpenGL with or without expanded visuals.

Note – In Solaris 2.5.1-based systems, an OpenGL-capable overlay visual is present only if you run `/usr/sbin/ffbconfig -sov` *enable* before the Window system is started. You must run this command as `root`.

The advantage to the overloading of X visuals is that the X server can be specific about the frame buffer configurations that the graphics hardware provides. This approach also enables the OpenGL implementation to better manage resources. Instead of allocating the maximal amount of resources for each window, the OpenGL implementation only needs to allocate the resources necessary for the GLX visual the application has selected. Thus, the application has more direct control over resource allocation.

Using the `glXGetConfig`(3gl) and `glXChooseVisual`(3gl) routines, applications can get information on the supported visuals and choose the appropriate visual. For helpful information on GLX programming, refer to *OpenGL Programming for X Windows Systems* by Mark Kilgard and *OpenGL Programming Guide*.

Table 4-1 lists OpenGL-capable visuals with expanded visuals.

Table 4-1 OpenGl-capable Visuals With Expanded Visuals

Double Buffer Capable?	GLX BufferSize	X Visual Class	GL_RGBA	Gamma Corrected?	GLX Level
Yes	24	TrueColor	True	No	0
Yes	24	TrueColor	True	Yes	0
Yes	24	DirectColor	True	No	0
Yes	8	PseudoColor	False	No	0
No	24	TrueColor	True	No	0
No	24	TrueColor	True	Yes	0
No	24	DirectColor	True	No	0
No	8	PseudoColor	False	No	0
No	8	PseudoColor	False	No	1

When the frame buffer video mode is monoscopic, only GL_MONO versions of these visuals are supported. In a stereoscopic video mode, both GL_MONO and GL_STEREO versions of these visuals are supported.

Table 4-2 lists OpenGL-capable visuals without expanded visuals.

Table 4-2 OpenGL-capable Visuals Without Expanded Visuals

Double Buffer Capable?	GLX BufferSize	X Visual Class	GL_RGBA	Gamma Corrected?	GLX Level
Yes	24	TrueColor	True	No	0
Yes	24	TrueColor	True	Yes	0
Yes	24	DirectColor	True	No	0
Yes	8	PseudoColor	False	No	0
No	8	PseudoColor	False	No	1

Colormap Flashing for OpenGL Indexed Applications

With the visuals exploded, there is greater potential for colormap flashing to occur for OpenGL indexed applications. This is because applications are forced to create private colormaps in order to create windows on the GLX visual they choose. In the Solaris 2.6 release, the colormap flashing problem is eased by the colormap equivalence feature. This feature allows OpenGL color indexed applications to be written in a way that creates less flashing.

Colormap equivalence allows a program to assign a colormap of one visual to a window that was created with a different visual, as long as the two visuals are colormap equivalent. This means, in general, that they share the same plane group and have the same number of colormap entries. The standard X11 protocol does not let programs mix visuals of colormaps and windows in this way. For more information on colormap equivalence, see the `XSolarisCheckColormapEquivalence`(3) man page.

Colormap equivalence is useful for OpenGL programs because the GLX visual expansion creates up to four different variants of each base `GL_CAPABLE` visual. So, for example, instead of one 8-bit default PseudoColor colormap, there may be a double-buffered variant, a stereo variant, and so on. Without colormap equivalence, an application cannot assign the default colormap to windows of these variant visuals, and this will result in more colormap flashing. With colormap equivalence, windows of all variants can share a colormap that was created using the base visual, and less colormap flashing will occur.

GL Rendering Model and X Visual Class

OpenGL RGBA rendering is supported on the 24-bit TrueColor and DirectColor visuals. OpenGL indexed rendering is supported on the 8-bit PseudoColor visuals and on the indexed or 224-color overlay visuals.

Depth Buffer

All GL-capable visuals, except for overlay visuals, have a 28-bit Z buffer (`GLX_DEPTH_SIZE == 28`).

Accumulation Buffer

All GL RGBA visuals have a (16, 16, 16, 16) accumulation buffer (`GLX_ACCUM_RED_SIZE == GLX_ACCUM_GREEN_SIZE== GLX_ACCUM_BLUE_SIZE == GLX_ACCUM_ALPHA_SIZE = 16`).

Stencil Buffer

All GL capable visuals, except for the overlay and stereo visuals, have a 4-bit stencil buffer (`GLX_STENCIL_SIZE == 4`).

Auxiliary Buffers

Auxiliary buffers are not supported by the Solaris OpenGL product (`GLX_AUX_BUFFERS == 0`).

Stereo

Note – This section is specific to Creator and Creator3D systems.

To run a stereo application in stereo mode, the frame buffer must be configured for stereo operation.

▼ To Set Up the Frame Buffer for Stereo Operation:

1. **Exit the window system.**
2. **Type this command:**
   For Solaris 2.5.1 HW297 `/usr/sbin/ffbconfig -res stereo -expvis` *`enable`*

   For Solaris 2.6 `/usr/sbin/ffbconfig -res stereo`
   Note that in the Solaris 2.6 release, this command must be run under superuser permissions or sys admin permissions.
3. **Restart the window system.**

Application can now use the stereo hardware buffers.

4

Rendering to DirectColor Visuals

The OpenGL API has no support for color mapping. The only way to get a DirectColor visual is to implement visual selection in the application using `XGetVisualInfo`(3gl) and `glXGetConfig`. If you request a visual with `glXChooseVisual`, you will get a 24-bit TrueColor visual for RGBA rendering and an 8-bit PseudoColor visual for index rendering.

When rendering to DirectColor visuals, the GL system calculates pixel values in the same way as it does for TrueColor visuals. The application is responsible for loading the window colormap with cells that make sense to the application.

Overlays

The Creator and Creator3D systems have one 8-bit overlay visual in monoscopic mode and two 8-bit overlays in stereo mode. The overlay visual GLX level is greater than zero (`GLX_LEVEL > 0`). Visuals with a GLX level less than or equal to zero are underlay visuals.

Server Overlay Visual (SOV) Convention

Server Overlay Visual (SOV) is an API for rendering transparent pixels in an overlay window. A transparent pixel is a special pixel code that allows the contents of underlay windows underneath to show through. SOV derives its name from the X property that informs the user of the special transparent pixel value: `SERVER_OVERLAY_VISUALS`. This value can be used as the input value to `glIndex*` calls so that the transparent pixel can be rendered into the overlay.

The SOV API, while not an X11 standard, is a convention that is supported by many X11 vendors. It is described at length in the book *OpenGL Programming for the X Window System* by Mark J. Kilgard. This section describes Sun-specific aspects of the SOV implementation.

Note – In this section, the term underlay is used as a synonym for the normal planes referred to in *OpenGL Programming for the X Window System*.

The `SERVER_OVERLAY_VISUALS` property describes visuals with transparent pixels (TransparentType = TransparentPixel), and also completely opaque visuals (TransparentType = None). If you need an overlay visual with a

transparent pixel, make sure that you check the TransparentType field of the entries in this property. The remainder of this section will discuss only the TransparentPixel SOV visuals.

Enabling SOV Visuals

SOV visuals are present by default in Solaris 2.6. But in Solaris 2.5.1 HW297, they must be explicitly enabled. SOV visuals can be enabled in an OpenWindows environment by becoming `root`, then typing the following command before starting the OpenWindows™ system: `/usr/sbin/ffbconfig -sov` *enable*`.` Then restart the Window system.

Both Creator and Creator3D platforms support SOV visuals. When these devices are configured for a monoscopic video mode, there is one TransparentPixel SOV visual. When in a stereoscopic video mode, there are two TransparentPixel SOV visuals exported: a monoscopic visual and a stereoscopic visual.

Note – Regardless of the video mode, there is always one overlay visual exported on these devices that is *not* SOV-capable. This visual is provided in order to support OVL, the Sun-specific overlay extension. This visual is not `GL_CAPABLE` and is never returned by `glXChooseVisuals`.

OpenGL Restrictions on SOV

Note – Creator and Creator3D systems do not directly support SOV, so the Solaris OpenGL 1.1 software provides the SOV support using a low-overhead software translation mechanism. If a program follows the restrictions described below, this mechanism provides rendering to SOV windows at full hardware speeds in most cases.

SOV is fully supported on SOV-capable visuals except for the following features, which are not supported:

- Uncorrelated window configurations. These window configurations are described below.
- Read back of transparent pixels via `glReadPixels`.
- Interframebuffer copies of transparent pixels via `glCopyPixels`.

- Logic operations other than `GL_COPY`.
- Index masks other than 0xff.
- `glShadeModel (GL_SMOOTH.`

If one of these unsupported features is used, rendering will complete without generating an error but the visual results will be undefined.

A correlated window configuration is a combination of an overlay and an underlay window that are the exact same size and shape. Typically, the overlay window is a child of the underlay window, but it may also be a sibling. In either case, there must be no other windows (mapped or unmapped) intervening between them. Once the window configuration is set up, it should not be changed by re-parenting one of the windows. If a window configuration doesn't meet this definition, then it is called an uncorrelated configuration and is not supported by OpenGL.

The application is responsible for maintaining the correlated relationship. The system does not maintain it automatically. The client must check for underlay window shape changes and if any occur, it must perform the equivalent changes on the overlay window.

Compatibility of SOV with other Overlay Models

Programs that use SOV visuals may coexist on the same screen with programs that use OVL, the Sun-specific overlay extension. But the two may not be used simultaneously with the same window.

Some XGL™ and OpenGL 1.0 programs are written to use the SOV transparent pixel if the SOV property is present, and to use XOR rendering in the default underlay visual if the SOV property is not present. These programs may not behave properly when the SOV property is present. When the SOV property is not present and the underlay is being used, a program may simply attach the default colormap to the default visual underlay window. In the presence of the SOV visual, the program will switch to using the SOV overlay visual but may continue to use the default colormap. Since the SOV overlay visual is usually not the same as the default visual, this will result in an X11 BadMatch error when the program attempts to attach the colormap to the overlay window. Care should be taken to write programs that always attach colormaps of the

proper visual to overlay windows. In this case, the program should have created a colormap using the SOV visual instead of trying to use the default colormap.

Programs that use SOV can also coexist with programs using the Sun visual overlay capability `glXGetTransparentIndexSUN`. However, `glXGetTransparentIndexSUN` is deprecated. It is provided only for compatability for existing programs that use it. Newly written transparent overlay programs should use SERVER_OVERLAY_VISUALS instead.

For information on using the Sun visual overlay capability, see the `glXGetTransparentIndexSUN` man page. In addition, look at the overlay example programs included in the `SUNWglut` package. These programs are installed by default into the directory `/opt/SUNWsdk/sdk_2.5/GL/contrib/examples/sun/overlay`.

Gamma Correction

On Creator and Creator3D workstations, two 24-bit TrueColor visuals are exported. One is gamma corrected; the other is not. To support imaging and Xlib applications, the nonlinear (not gamma-corrected) visuals are listed before linear visuals. However, to provide linear visuals for graphics applications running under the Solaris OpenGL software, the `glXChooseVisual()` call was modified to return a linear visual.

If you want to use a nonlinear TrueColor visual, you need to get the visual list from Xlib. Use the Solaris API `XSolarisGetVisualGamma`(3) to query the linearity of the visual. To determine whether a visual supports OpenGL, call `glXGetConfig` with *attrib* set to `GLX_USE_GL`.

If you are using another vendor's OpenGL and displaying your application on a Creator or Creator3D graphics workstation, and you want to use a linear visual, run the command `/usr/sbin/ffbconfig -linearorder first` to change the order of visuals so that the linear (gamma-corrected) visual is the first visual in the visual list. See *Solaris X Window System Developer's Guide* for more information on gamma correction and `XSolarisGetVisualGamma`.

4

Tips and Techniques 5

This chapter presents miscellaneous topics that you may find useful as you port your application to the Solaris OpenGL software.

Identifying the Solaris OpenGL Library Version

You can identify the library version number and build date of the OpenGL library components (`libGL.so`, `libGLU.so`, `libGLw.so`, `SUNWGLX.so`) using the `what(1)` command. For example, any of the following commands:

```
% what /usr/openwin/lib/libGL.so
% what /usr/openwin/lib/libGLU.so
% what /usr/openwin/lib/libGLw.so
% what /usr/openwin/server/modules/SUNWGLX.so.1
```

would result in something like:

```
RELEASE VERSION: SUNWglrt Solaris OpenGL version 1.1, libGL.so.1,
sparc, [build date]
RELEASE VERSION: SUNWglrt Solaris OpenGL version 1.1, libGLU.so.1,
sparc, [build date]
RELEASE VERSION: SUNWglwrt Solaris OpenGL version 1.1, libGLw.so.1,
sparc, [build date]
RELEASE VERSION: SUNWglrt Solaris OpenGL version 1.1, SUNWGLX.so.1,
sparc, [build date]
```

This identifies the OpenGL component to be Solaris OpenGL version `1.1` built on the specified build date.

5

Avoiding Overlay Colormap Flashing

Colormap flashing may occur when your application uses overlay windows. This problem stems from several characteristics of the Creator3D system: the overlay visual is not the default visual, the Creator3D is a single hardware colormap device, and X11 allocates colormap cells from pixel 0 upward. When the application renders to the overlay window, it must use a non-default visual, and a non-default colormap is loaded. In this case, colormap flashing between the default and non-default colormaps can occur.

The best solution to this problem is to allocate the overlay colors at the high end of the overlay colormap. In other words, if you have *n* colors to allocate, allocate them in the positions *colormap_size -n -1* to *colormap_size -1*.This avoids the colors in the default colormap, which are allocated upward starting at 0. To allocate *n* colors at the top of the overlay colormap, first allocate *colormap_size-n* read / write placeholder cells using `XAllocColorCells`. Then allocate the *n* overlay colors using `XAllocColor`. Finally, free the placeholder cells. This solution is portable; it works on both single- and multiple-hardware colormap devices.

Changing the Limitation on the Number of Simultaneous GLX Windows

There is a limitation on the number of GLX windows that an application can use simultaneously. Each GLX window that has an attached GLX context uses a file descriptor for DGA (Direct Graphics Access) information. You can find the current number of open file descriptors using the `limit`(1) command:

```
% limit descriptors
descriptors 64
```

The system response tells you that you have up to 64 direct GLX contexts, assuming that you have no other processes concurrently using file descriptors.

You can increase the per-process maximum number of open file descriptors using the `limit` command as follows:

```
% limit descriptors 128
```

This command changes the number of file descriptors available for DGA and other uses to 128. Use the `sysdef`(1M) command to determine the maximum number of file descriptors for your system.

Hardware Window ID Allocation Failure Message

On Creator3D, when a program calls `glXMakeCurrent`(3gl) to make a window the current OpenGL drawable, the system will attempt to allocate a unique hardware window ID (WID) for the window. This allows double buffering and hardware WID clipping to be used. Because hardware WIDs are a scarce resource and can be used for other purposes, there might not be any WIDs available when `glXMakeCurrent` is called. If this should happen, the following message is displayed:

```
OpenGL/FFB Warning: unable to allocate hardware window ID
```

In this situation, double buffering will not be provided for the window, and the window will be treated as a single-buffered window.

Getting Peak Frame Rate

The frame rate that `ogl_install_check` prints out is symchronized to monitor frequency. It measures the time it takes to render the frame, wait for `vblank`, then swap the buffers. Since `FFB` can render the `ogl_install_check` image very quickly, even on an `FFB1` Electron 167 mhz machine, the bottleneck is waiting for the monitor `vblank`. So, under normal circumstances, `ogl_install_check` is never going to be able to get a frame rate faster than the monitor frequency.

However, there is an environment variable called `OGL_NO_VBLANK` that you can set to see the peak, unsynchronized frame rate. When set, this environment variable swaps buffers immediately, without waiting for `vblank`.

Frequently Asked Questions

How can I find out the Release Version Number of the OpenGL Library I am using?

You can identify the Release Version Number of the OpenGL Library by:

1. Using the `what`(1) or `mcs`(1) command:

```
% what /usr/openwin/lib/libGL.so.1
```

```
% mcs -p /usr/openwin/lib/libGL.so.1
```

2. Programatically, by calling `glGetString (GL_VERSION)`

(see the `glGetString` man page for more details)

3. Running the Solaris OpenGL `install_check` demo program:

```
% /usr/openwin/demo/GL/ogl_install_check
```

What is the maximum number of GLX windows that can be used simultaneously?

Each window that is created and to which a GLX context is attached uses a file descripter of Direct Graphics Access (DGA) information. The per-process maximum number of open file descriptors can be found, and changed, using the `filename` command:

```
% limit descriptors

descriptors 64
```

This implies you have up to 64 direct GLX contexts (assuming you have other things in that process that use up file descriptors).

This limit can be increased by typing:

```
% limit descriptors 128
```

This will increase the number of file descriptors available for DGA use and other uses to 128. The `sysdef`(1M) command will tell you what the maximum number of file descriptors is, along with other information.

In addition to the limit on descriptors, there is a limit on the number of hardware double-buffered windows. On Creator3D, you have up to 32 double-buffered windows. Beyond that, OpenGL defaults to single-buffered mode.

The Creator3D graphics accelerator supports only double-buffered visuals. I want to use single-buffer behavior. How can I do this?

You can simulate single-buffer behavior using double-buffered context by calling `glDrawBuffer (GL_FRONT)` and avoiding calls to `glXSwapBuffers.`